

2. Automata and Regular Expressions

Contents

- Regular expressions
- Generalized NFSA (Transition graphs)
- Kleene's Theorem

Regular expressions

- **Regular languages** comprise the class of all languages recognisable by FSA (by definition).
- Regular languages may also be described by **regular expressions** (RE). (We will prove this today.)
- **Definition:** The class of REs for a given alphabet Σ is defined inductively by
 - Every letter of Σ is a RE.
 - ε , the empty string, is a RE.
 - $\emptyset$ is a regular expression
 - If r and s are RE, then so are
 - (r) , rs , $r \cup s$, r^* .
 - Nothing else is a RE.

Regular expressions

- Concatenation $R \circ S$ (or just RS)
 - A RE is a set of strings. For sets (and hence for REs) concatenation is defined as:

$$R \circ S = \{rs \mid r \in R \text{ and } s \in S\}$$

- Union $R \cup S$

$$R \cup S = \{r \mid r \in R \text{ or } r \in S\}$$

- Kleene's star: R^*

$$R^* = \{r_1 r_2 \dots r_k \mid k \geq 0 \text{ and each } r_i \in R\}$$

- ex.: $R = \{ab, ba\}$ $abba$ or $ababbaab$ are in R^* , aab is not in R^*
- Note: $k \geq 0$ includes $k = 0$, so the empty string $\mathcal{E}$ is always in R^*

Regular expressions

- Concatenation $R \circ S = \{rs \mid r \in R \text{ and } s \in S\}$

$\varepsilon \circ \varepsilon = \varepsilon$ remember that ε is the absence of a symbol

$$R \circ \varepsilon = \varepsilon \circ R = R$$

$$R \circ \emptyset = \emptyset \circ R = \emptyset$$

- Union $R \cup S = \{r \mid r \in R \text{ or } r \in S\}$

$R \cup \varepsilon = \varepsilon \cup R$ is different than R if R does not include the empty string

$$R \cup \emptyset = \emptyset \cup R = R$$

- Kleene's star: R^* $R^* = \{r_1 r_2 \dots r_k \mid k \geq 0 \text{ and each } r_i \in R\}$

$\varepsilon^* = \{\varepsilon\}$ since concatenation of empty is empty

$\emptyset^* = \{\varepsilon\}$ with $k=0$ concatenations

Example

- The RE $0 \cup 1(0 \cup 1)^*$ describes (or defines) a language (i.e., a set of words) in which every word is either
 - the symbol 0, all by itself, or
 - the symbol 1 followed by any sequence (including the null sequence) of 0s and 1s, in any order.

Precedence of operations

- $R^* > \text{concatenation} > \text{union}$
- This precedence is implicit if parenthesis are not used

$0 \cup 1(0 \cup 1)^* = 0 \cup (1(0 \cup 1)^*)$ but different than $(0 \cup 1)(0 \cup 1)^*$

Use parenthesis to avoid errors and that what you write is not what you think!

More examples

$$\Sigma = \{0, 1\}$$

- $0^*10^* =$
- $\Sigma^*1\Sigma^* =$
- $(0 \cup \varepsilon)1^* = 01^* \cup 1^*.$
- $1^*\emptyset = \emptyset.$

More examples

$$\Sigma = \{0, 1\}$$

- $0^*10^* = \{w \mid w \text{ contains a single } 1\}.$
- $\Sigma^*1\Sigma^* =$
- $(0 \cup \varepsilon)1^* = 01^* \cup 1^*.$
- $1^*\emptyset = \emptyset.$

More examples

$$\Sigma = \{0, 1\}$$

- $0^*10^* = \{w \mid w \text{ contains a single } 1\}.$
- $\Sigma^*1\Sigma^* = \{w \mid w \text{ has at least one } 1\}.$
- $(0 \cup \varepsilon)1^* = 01^* \cup 1^*.$
- $1^*\emptyset = \emptyset.$

Variables

- Intermediate variables may be freely introduced into the equations of REs, as in the following example:
 - The RE for the language of Nonnegative Integers is
 - $NI = 0 \cup N(N \cup 0)^*$
 - $N = 1 \cup 2 \cup 3 \cup 4 \cup 5 \cup 6 \cup 7 \cup 8 \cup 9$
- Read as: *Every word in the language Nonnegative Integers is either the symbol '0', all by itself, or a word of the language N followed by any sequence (including the null sequence) of words from the language N or '0's, in any order, where every word in the language N is either '1', or '2',... or '9'.*

Regular Expressions and Languages

- **Definition:** The language described by an RE is defined inductively by:
 - If r is a RE with $r = s \in \Sigma$ (i.e. a single letter), then r describes the language $\{s\}$. This language contains only one word and that word contains one letter.
 - If r is a RE describing the language L and s a RE describing M , then rs describes the language LM , the set of words each of which is a word in L concatenated with a word in M .

Regular Expressions and Languages (cont.)

If r is a RE describing the language L , and s a RE describing M
– $r \cup s$ describes the language $L \cup M$, the set of words each of which is either in L or in M .

– r^* is the **Kleene closure (or star)** of the language L (that is, the language of L^*), defined by

- $L^* = \{x_1x_2\dots x_n \mid x_i \in L \text{ and } n \geq 0\}$
($n = 0$ gives the null string ε).

The regular expression ε describes $\{\varepsilon\}$

(Note: concatenation, union, and star are the so-called regular operations on languages)

REs, NFSA and Lexical Analysis

- When writing compilers/interpreters the first step is to write code that can read in a program and divide it into tokens of the programming language (keywords, numbers, strings, identifiers, etc)
- Typically tokens are specified using REs
- As we will see shortly, these REs can be converted naturally into NFSA (and then DFSA) which will consume the program as input

Kleene's theorem

- Any language that can be defined by
 - regular expressions (RE), or
 - finite state automata (FSA), or
 - Generalized NFSA (GNFSA),can be defined by all three methods.

We need to prove:

$RE \Rightarrow FSA$

$FSA \Rightarrow RE$

We will introduce GNFSAs as part of the 2nd proof

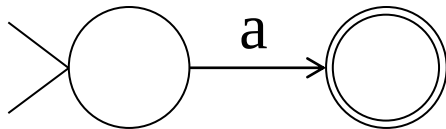
Proof $RE \subseteq FSA$

- This proof relies on the recursive definition of REs and provides a FSA construction algorithm
 - **Definition:** The class of REs for a given alphabet Σ is defined inductively by
 - Every letter of Σ is a RE.
 - ε , the empty string, is a RE.
 - $\emptyset$ is a regular expression
 - If r and s are RE, then so are
 - (r) , rs , $r \cup s$, r^* .
 - Nothing else is a RE.
- The construction algorithm provides a graphical counterpart for each of the regular operations

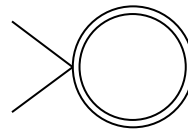
Proof $RE \subseteq FSA$

- Rule 1:** There is an FSA that accepts any particular letter of the alphabet. There is an FSA that accepts only the empty string. There is an FSA that recognizes $\emptyset$.

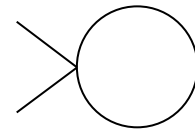
$$R = a \quad a \in \Sigma \quad L(R) = \{a\}$$



$$R = \varepsilon \quad L(R) = \{\varepsilon\}$$



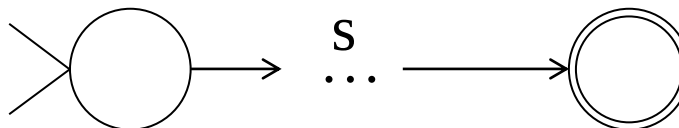
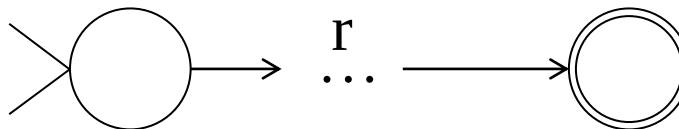
$$R = \emptyset \quad L(R) = \{\}$$



Note: we use non-deterministic FSAs

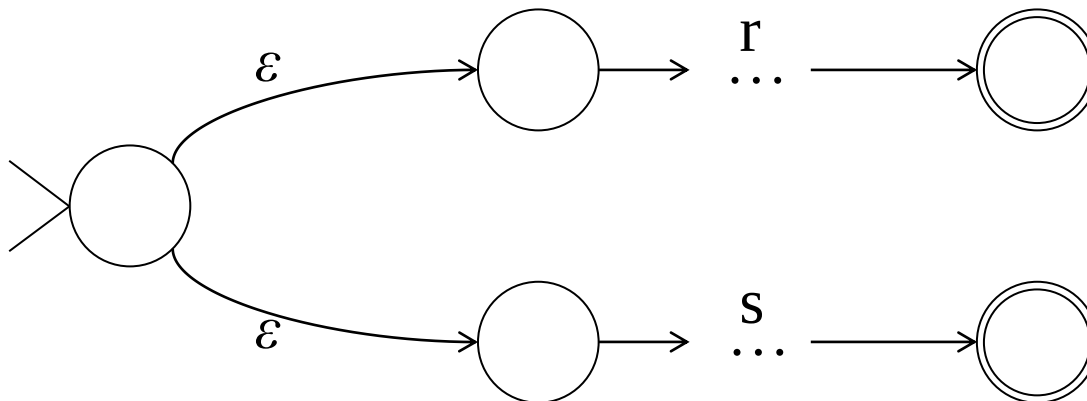
Proof $RE \subseteq FSA$

- **Rule 2:** If there is an FSA that accepts the language defined by the RE r and there is an FSA that accepts the language defined by the RE s then there is an FSA that accepts the language defined by the RE $r \cup s$.



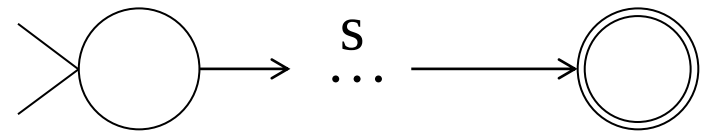
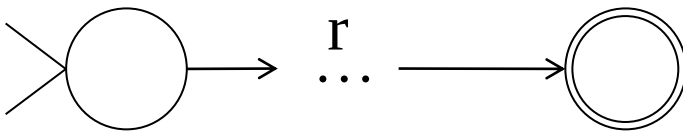
Proof $RE \subseteq FSA$

- Rule 2:** If there is an FSA that accepts the language defined by the RE r and there is an FSA that accepts the language defined by the RE s then there is an FSA that accepts the language defined by the RE $r \cup s$.



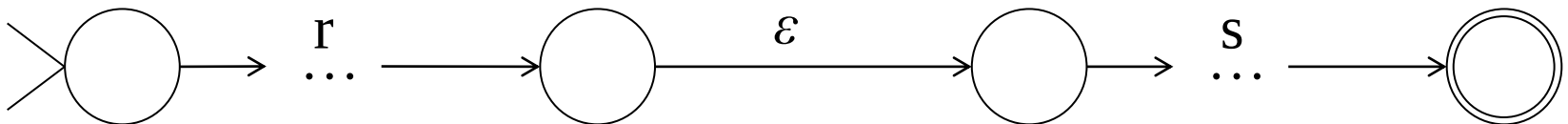
Proof $RE \subseteq FSA$

- **Rule 3:** If there is an FSA that accepts the language defined by the RE r and there is an FSA that accepts the language defined by the RE s then there is an FSA that accepts the language defined by the RE rs .



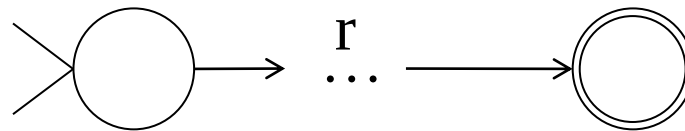
Proof $RE \subseteq FSA$

- **Rule 3:** If there is an FSA that accepts the language defined by the RE r and there is an FSA that accepts the language defined by the RE s then there is an FSA that accepts the language defined by the RE rs .



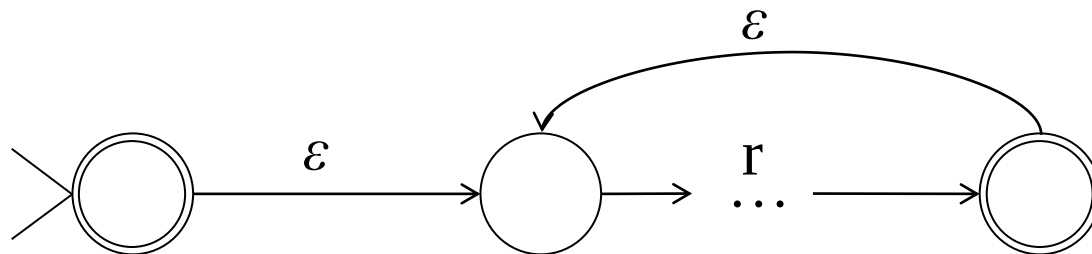
Proof $RE \subseteq FSA$

- **Rule 4:** If there is an FSA that accepts the language defined by the RE r then there is an FSA that accepts the language defined by the RE r^* .



Proof $RE \subseteq FSA$

- Rule 4:** If there is an FSA that accepts the language defined by the RE r then there is an FSA that accepts the language defined by the RE r^* .



End of Proof

Kleene's theorem

- Any language that can be defined by
 - regular expressions (RE), or
 - finite state automata (FSA), or
 - Generalized NFSA (GNFSA),can be defined by all three methods.

We need to prove:

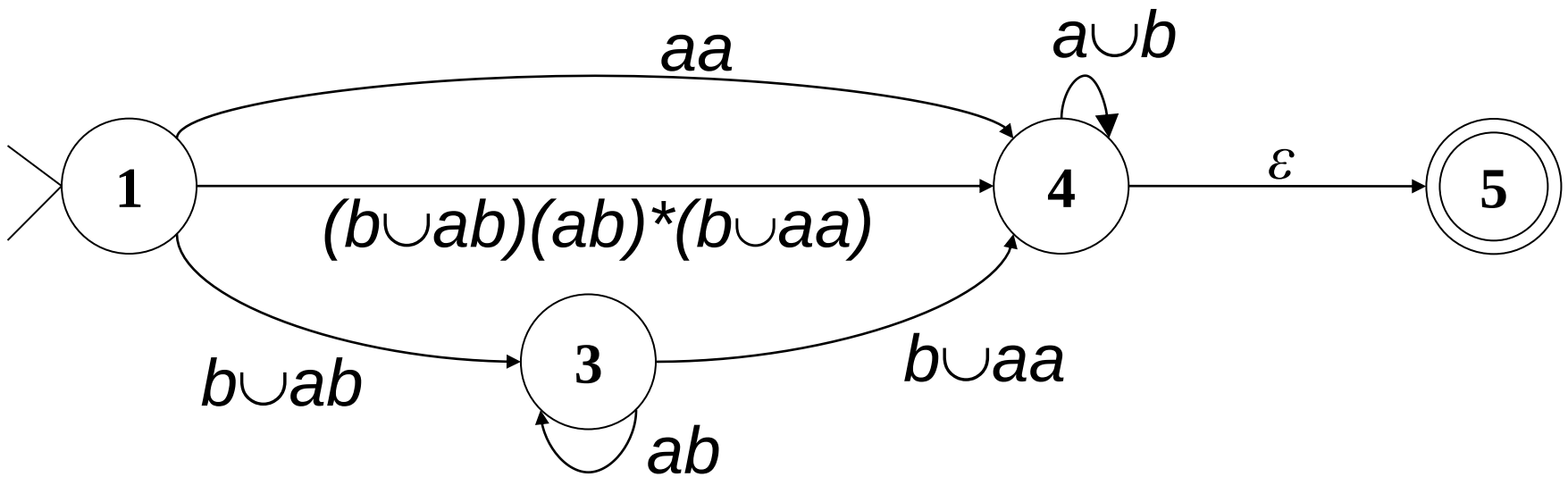
RE $\Rightarrow$ FSA ✓

FSA $\Rightarrow$ RE

Generalized NFSA

- A further extension to FSAs can be made that does not add to their power.
- A **generalized nondeterministic finite automata (GNFA)** is similar to a transition diagram but may have the following additional attribute:
 - Any edge may be labelled by a *regular expression*, including the null string, ϵ .
- A GNFA may make a state transition along such an arc if the string labelling that arc is a **prefix** of the current input string. (I.e., it reads a block of symbols from the input that is in the language described by the regular expression on the arc.)

GNFSA example



Terminological note

- Note that terminology varies: in a number of other books **generalized nondeterministic finite automata (GNFA)** are called **transition graphs (TG)**.
- Expression diagram is another name sometimes used.
- Same concept, different names.

Generalized NFSA

- If an edge from a state is labelled with the null string, then the GNFA in that state may choose **non-deterministically** to make a state transition along that edge, like in NFSAs. It therefore does not add power.
- If an edge is labelled with the empty set, that transition can never be used and can be removed.
- Any edge of a GNFA that is labelled by a regular expression can be expanded into a sequence of transitions, one for each symbol in the string, and loops for $*$. This can be done analogously to the construction algorithm described above to convert REs to NFSAs.
- We conclude that the every language recognised by a GNFA is also recognisable by a FSA. (GNFA $\subseteq$ FSA)

$FSA \subseteq GNFA$

- Every finite state automaton is representable by a corresponding transition diagram (which is a GNFA). Therefore any language that has been defined by a FSA has already been defined by a GNFA.
- In other words, a FSA is a GNFA in which all REs in the transitions are symbols of the alphabet or the empty string.

Kleene's theorem

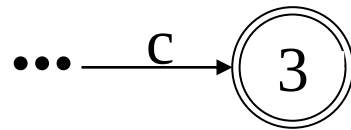
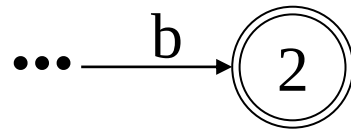
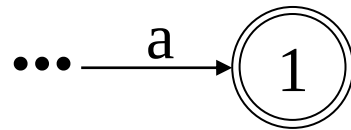
- The **proof** is in three parts:
 - 1. Every language that can be defined by a FSA (i.e., a regular language) can also be defined by a GNFA. ✓
 - 2. Every language that can be defined by a RE can also be defined by a FSA (thus it is regular). ✓
 - 3. Every language that can be defined by a GNFA can also be defined by a RE (thus, from 1., every regular language can be defined by a RE).
- In other words: A language is regular if and only if some regular expression describes it.

Proof $\text{GNFA} \subseteq \text{RE}$

- The proof of this part provides a **constructive algorithm** that will transform every conceivable GNFA into an equivalent RE **in a finite number of steps**.

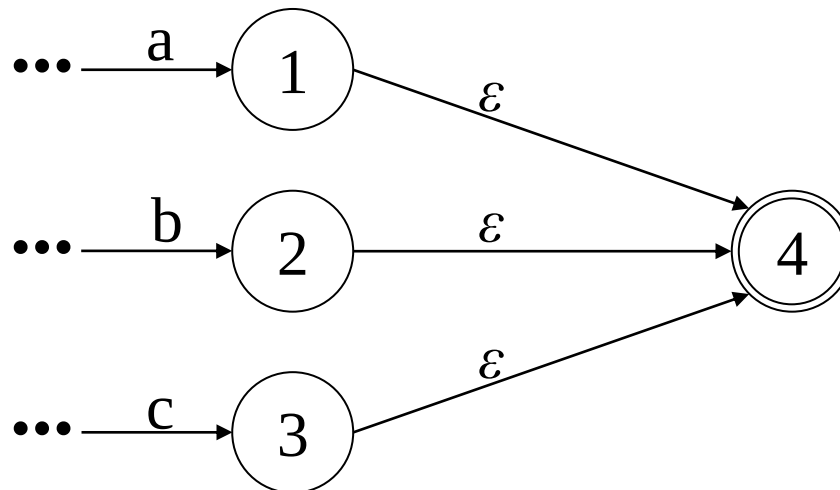
Step 1: Single Accept State

- Transform a GNFA with **more than one accept state** by defining a **new accept state** to which a ε -**transition** is constructed **from each** of the original accept states (which are no longer accept states).



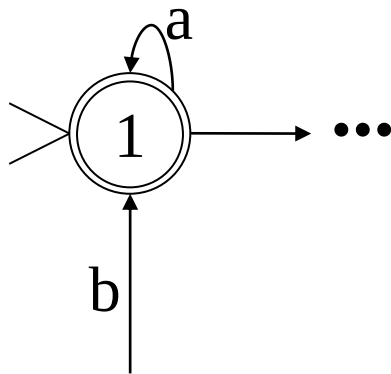
Step 1: Single Accept State

- Transform a GNFA with **more than one accept state** by defining a **new accept state** to which a ε -**transition** is constructed **from each** of the original accept states (which are no longer accept states).



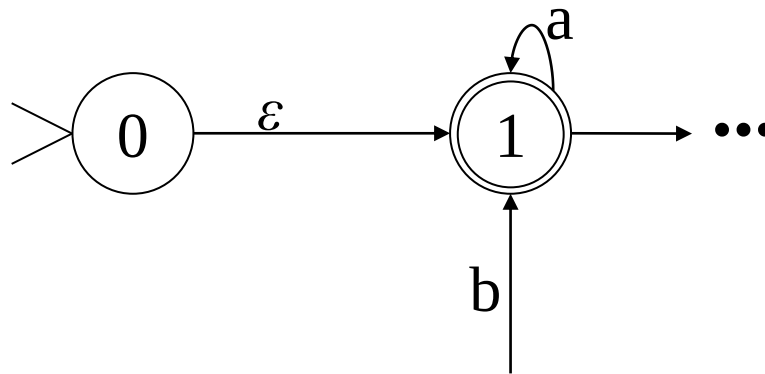
Step 2: No Transitions into Start State and Start State no Accept State

- Transform a GNFA with **one or more transitions into its start state or being an accept state** by defining a **new start state** from which a ε -**transition** is constructed to the original start state.



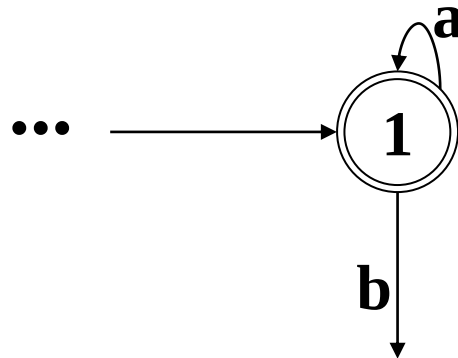
Step 2: No Transitions into Start State and Start State no Accept State

- Transform a GNFA with **one or more transitions into its start state or being an accept state** by defining a **new start state** from which a ε -**transition** is constructed to the original start state.



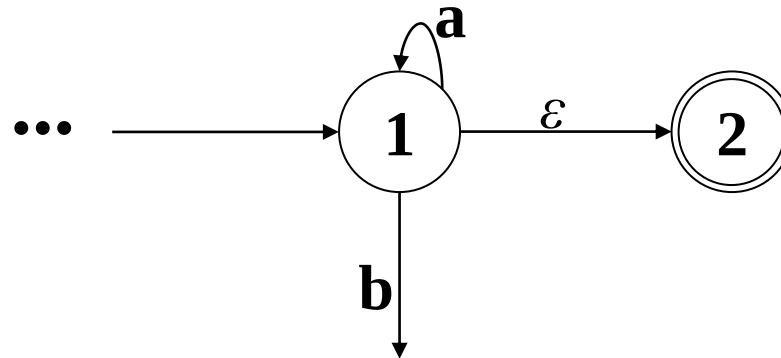
Step 3: No Transitions out of Accept State

- Transform a GNFA with **one or more transitions out of its accept state** by defining a **new accept state** to which a ε -transition is constructed **from** the original accept state from which the accept state signifier is **removed**.



Step 3: No Transitions out of Accept State

- Transform a GNFA with **one or more transitions out of its accept state** by defining a **new accept state** to which a ε -transition is constructed **from** the original accept state from which the accept state signifier is **removed**.



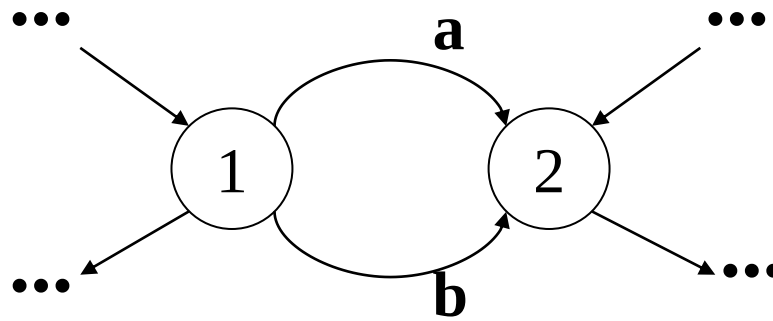
Step 1-3:

- A single Accept state nonoverlapping with the Start state
- No incoming transitions to the Start state
- No outgoing transitions from the Accept state

Note: States added should all have names different than existing states

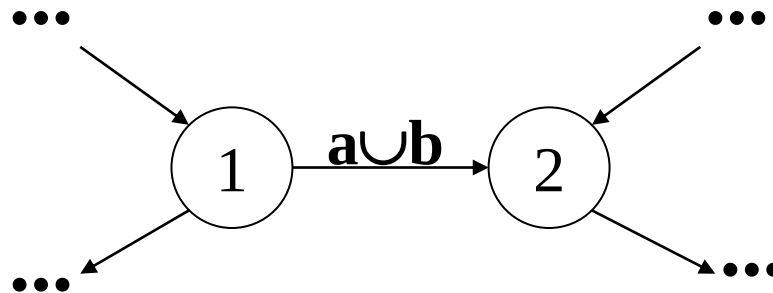
Step 4a: Sum Edges

- Any collection of edges (i.e. transitions) connecting the **same two states** in the **same direction** may be replaced by a **single** edge in that direction between those states labelled by the **union** of the labels on the edges it replaces.



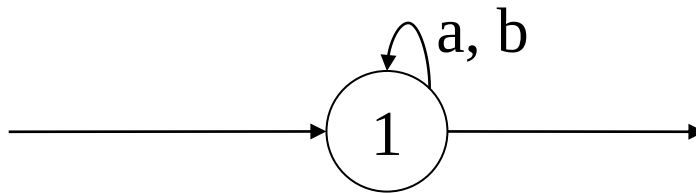
Step 4a: Sum Edges

- Any collection of edges (i.e. transitions) connecting the **same two states** in the **same direction** may be replaced by a **single** edge in that direction between those states labelled by the **union** of the labels on the edges it replaces.



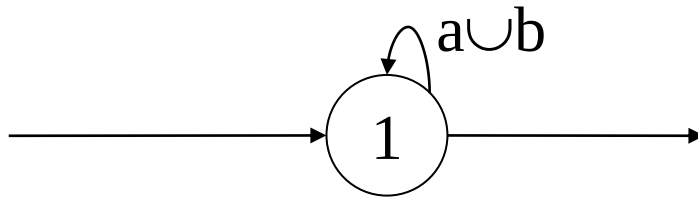
Step 4b: Sum Loops

- Since a **loop** on a state is just an edge from that state to itself, any collection of loops on the same state may be replaced by a single loop labelled by the union of the labels on the loops it replaces.



Step 4b: Sum Loops

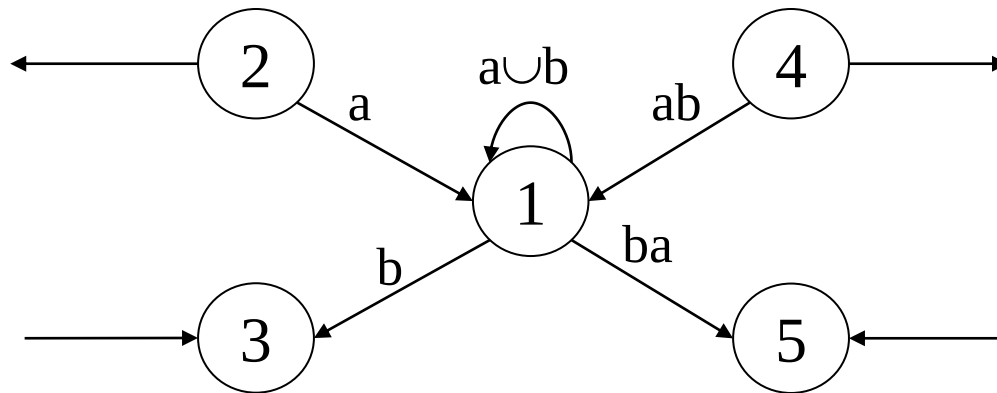
- Since a **loop** on a state is just an edge from that state to itself, any collection of loops on the same state may be replaced by a single loop labelled by the union of the labels on the loops it replaces.



Step 5: Bypass States

- Each application of the bypass operation removes **one state** (not the start or the accept ones) from a GNFA, at the same time transforming it into an **equivalent** GNFA, that recognises **exactly** the same language. Any order of bypass of states will do.
- The operation **restores every path broken** by the removal of the state.
- It should be **applied in stages**, which will be demonstrated by **bypassing state 1** in the following fragment of GNFA.

Step 5: Bypass States



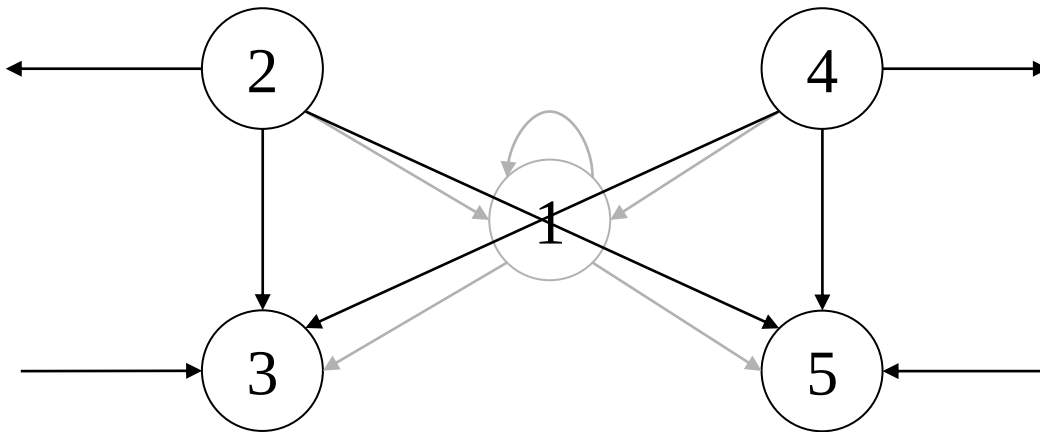
- Bypassing state 1:
- Identify the source of every **incoming** edge (here, 2 and 4) and the destination of every **outgoing** edge (here, 3 and 5) of the bypassed state, **excluding loops** on that state.

Step 5: Bypass States

- Record the **labels** on the incoming and outgoing edges and, if the state has a **loop** on it, record the label on the loop.
 - From 2, label a. From 4, label ab.
 - To 3, label b. To 5, label ba.
 - Loop, label $a \cup b$.
- Next **delete** the bypassed state and **all** its incoming and outgoing edges, but **remember the labels**.

Step 5: Bypass States

- **For every pair** comprising an incoming and an outgoing (to and from the deleted state 1) edge, add a new edge between its incoming ('from') and outgoing ('to') states, **in that direction**.

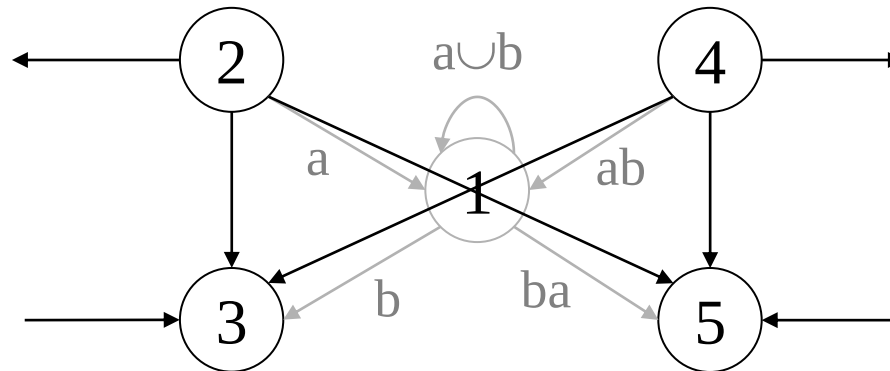


Cross check:

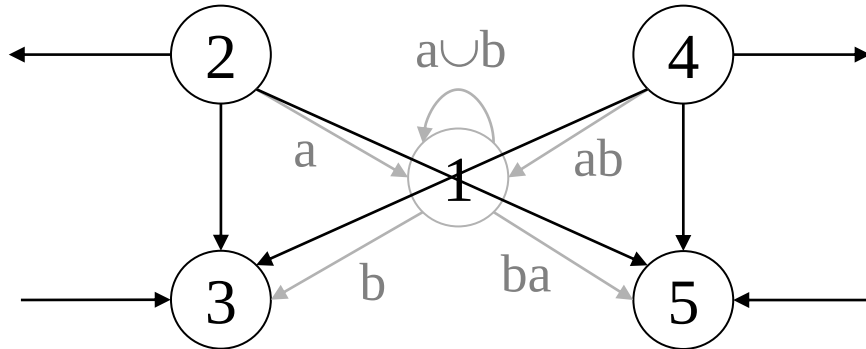
If there were m 'from' states and n 'to' states, then there should be $m \cdot n$ new edges. Here $m=2$, $n=2$, and there are exactly $2 \cdot 2 = 4$ new edges.

Step 5: Bypass States

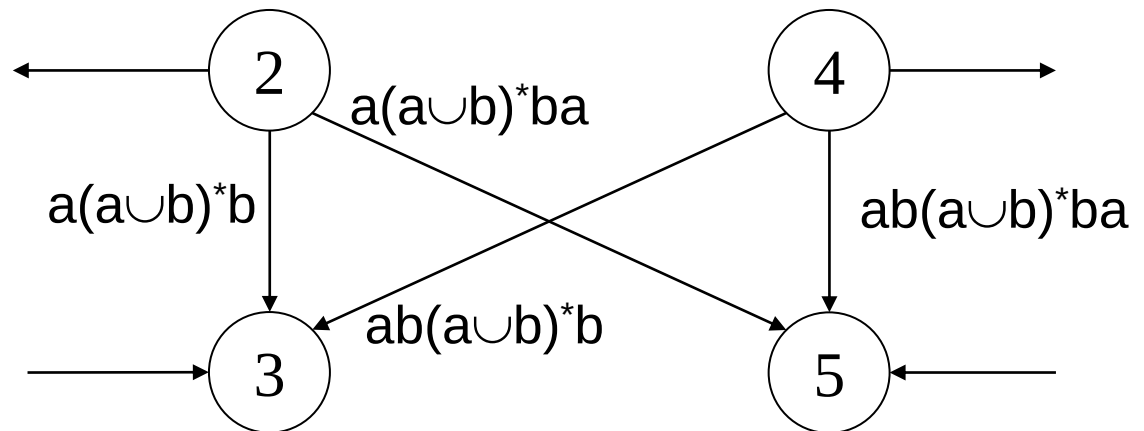
- **For each new edge, concatenate** the incoming label, the **Kleene Star** of the loop label (if there was one) and the outgoing label, **in that order**, and decorate the new edge with that expression.



Step 5: Bypass States



From	To	New Label
2	3	$a(a \cup b)^*b$
2	5	$a(a \cup b)^*ba$
4	3	$ab(a \cup b)^*b$
4	5	$ab(a \cup b)^*ba$



Iteration

- Steps 4 and 5 are iterated until no further transformation is possible.
- After performing each bypass, **do all valid sums on edges and loops (step 4) as there may now be parallel edges**, then choose the next state to bypass (step 5).
- This choice is completely arbitrary. Different sequences of bypass will produce different, but equivalent, regular expressions.
- Steps 1 to 3 should no longer be applicable (we don't touch start and accept states).
- Take care when doing concatenation and sums.
- **Use parentheses wherever necessary to avoid ambiguity.**

Termination

- As each bypass operation reduces the number of states in the GNFA by one, eventually there will be exactly **two** states left – the start state (created in step 2) and the accept state (created in step 1 or 3) – as no other steps can alter them.
- There must be exactly one transition from the start state to the accept state, because if there were two in the same direction, they would be summed by step 4.
- There can't be a transition from the accept state to the start state because of steps 1-3.

Termination

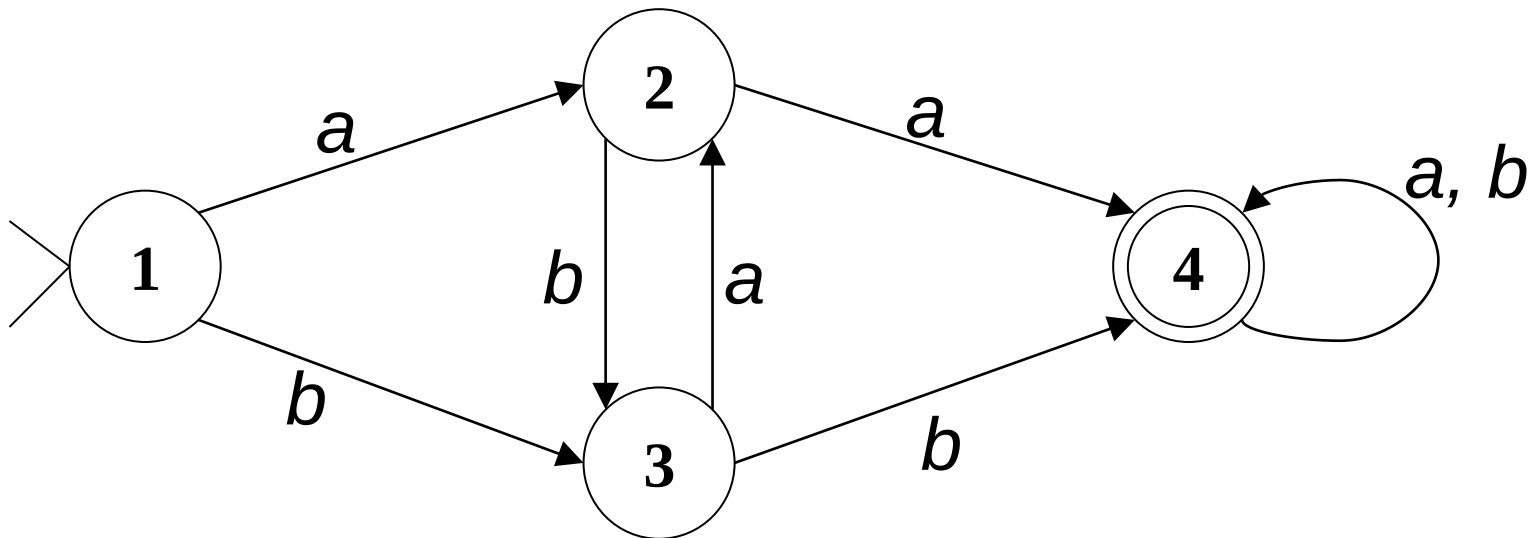
- All, and only, the operators of regular expressions are generated by these transformations: ' $\cup$ ' by step 4; concatenation and '*' by step 5, and parentheses wherever necessary.
- Each bypass generates an equivalent GNFA, so the reduced GNFA with only start and accept state is equivalent to the original.
- The result must therefore be a single regular expression labelling a single edge, and this regular expression must define the language accepted by the GNFA.
- **End of Proof.**

States with no outgoing edges

- **Note:** if a state has no outgoing edges, then when ‘bypassed’ it will simply be removed, along with any incoming edges.
- This is because following these edges cannot lead to a successful computation.

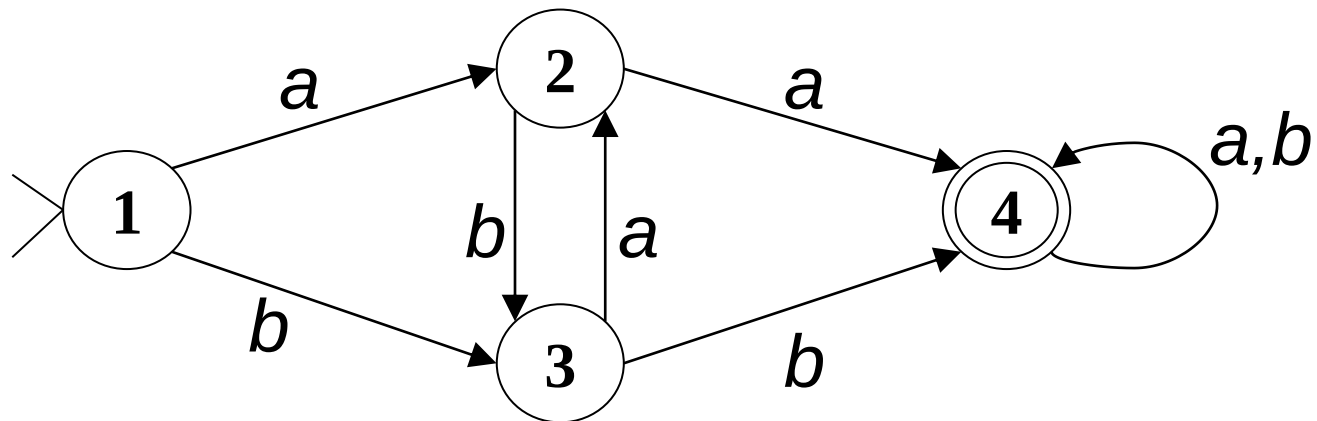
Applying Kleene's Theorem

- Derive a Regular Expression for the language recognised by this transition (state) diagram (graph):



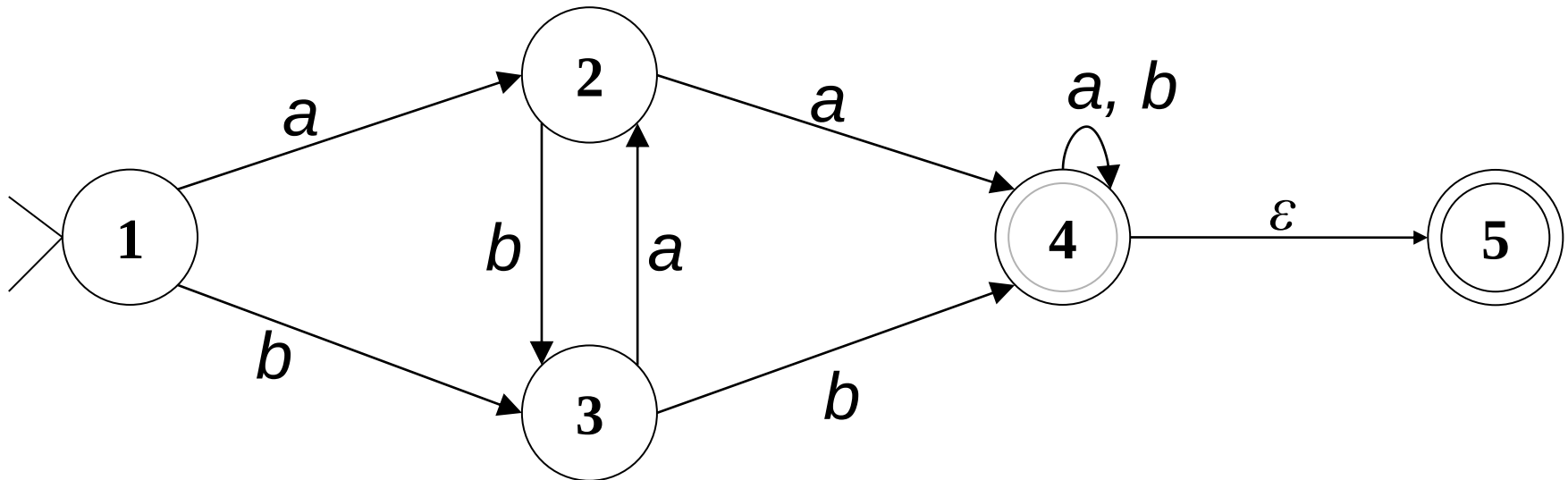
Example

- The graph already has exactly one accept state (4), so Step 1 is not applicable.
- The start state (1) has **no** transitions **into** it so Step 2 is not applicable.
- The FSA has one accept state (4) with **two** transitions **out** of it, labelled 'a' and 'b', both looping back to state 4, so Step 3 is applicable.



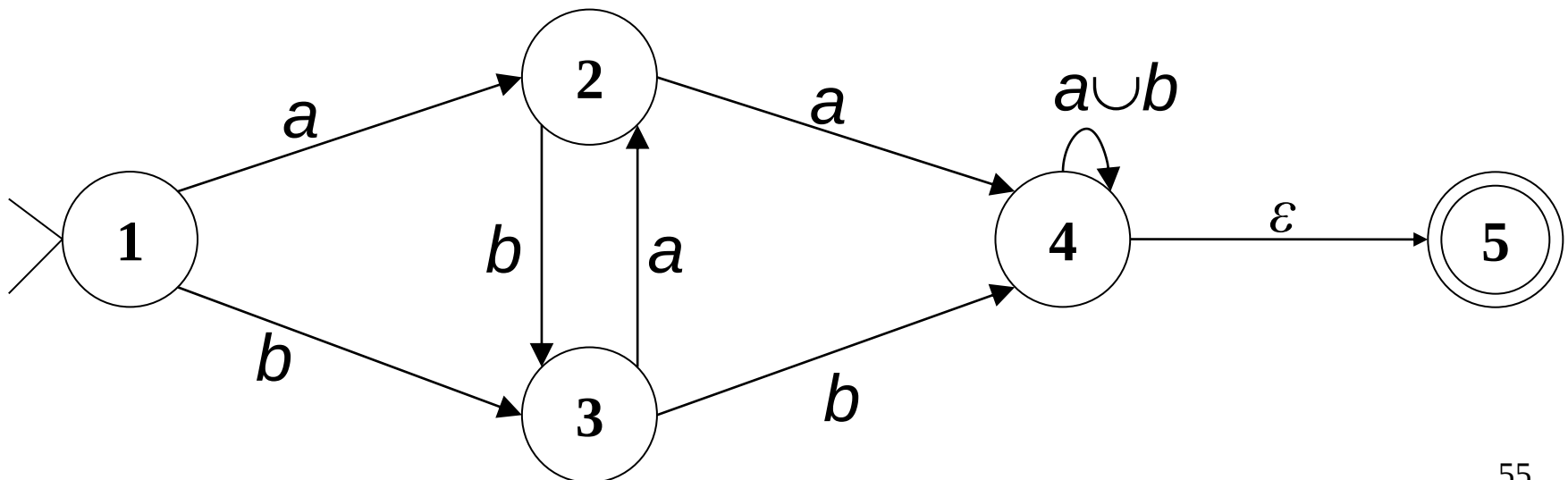
Example

- Apply the transformation in Step 3 by **adding** a new accept state (5), with a null transition to it from state 4, and **removing** the accept state marker from state 4.



Example

- There are two 'loops' on state 4, labelled 'a' and 'b', respectively. These are 'summed' by replacing them by a single loop, labelled with ' $a \cup b$ ' (Step 4b).
- (Note: the edges between states 2 and 3 are **not** in the same direction, so they **cannot** be summed. So Step 4a is not applicable.)



Example

- The only transformation now available is state bypass (Step 5).
- Any state may be chosen for bypass.
- Here, choose **state 2**.

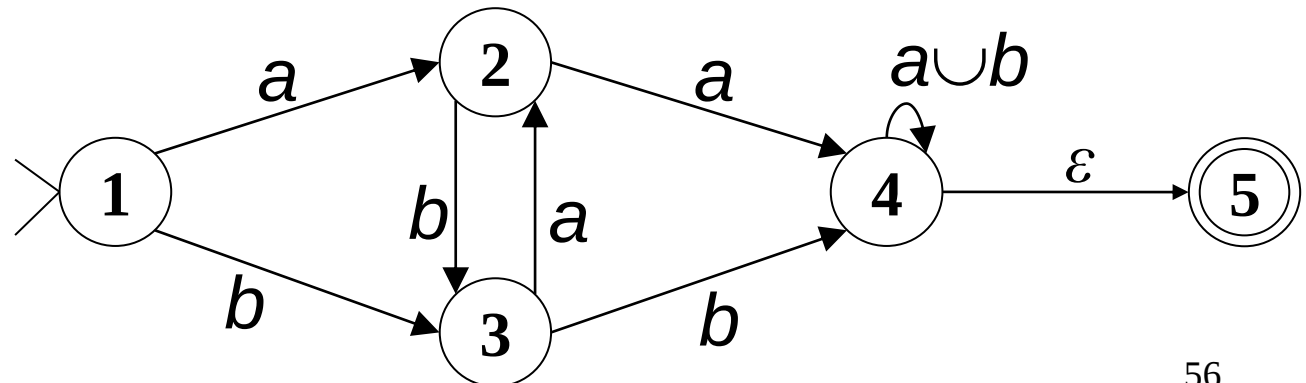
- Incoming Edges:**

From	Label
1	a
3	a

- Outgoing Edges:**

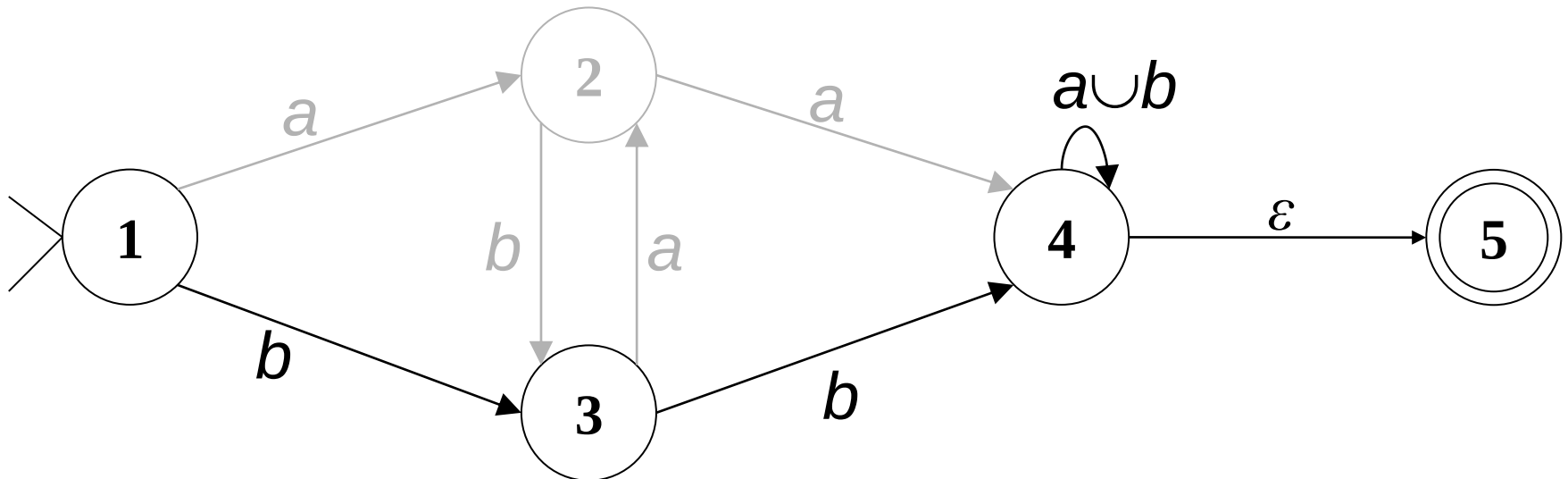
To	Label
3	b
4	a

- Loop Label:** None



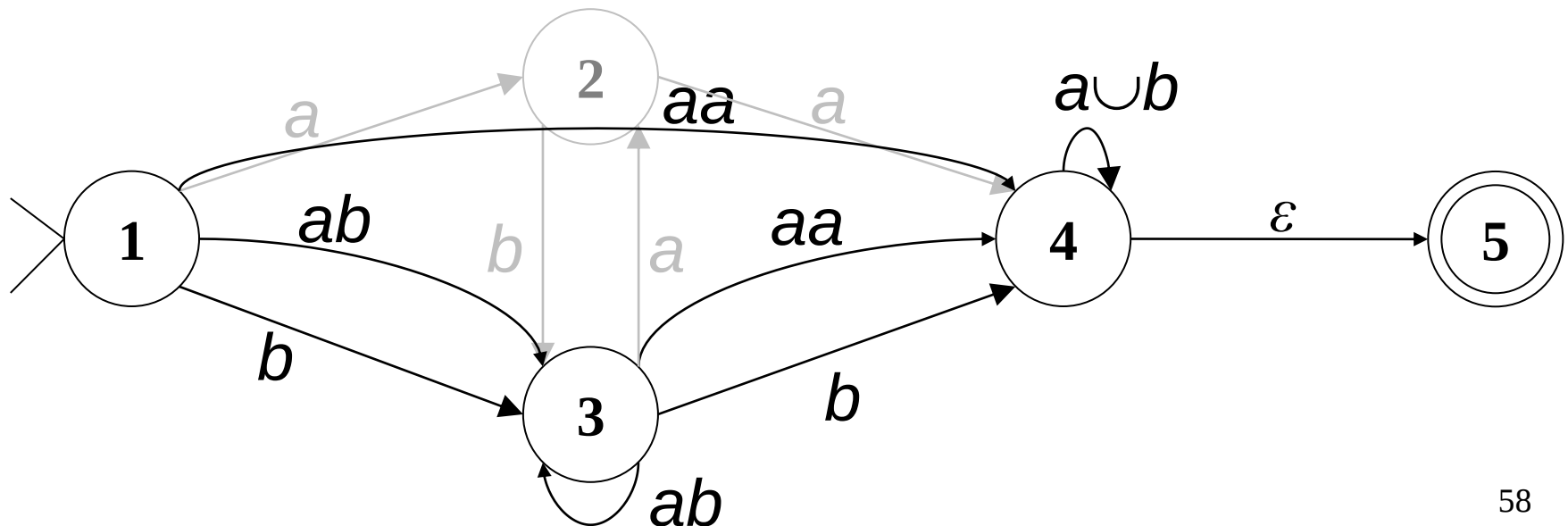
Example

- **Delete** the bypassed state and **all** its incoming and outgoing edges, but **remember the labels**.



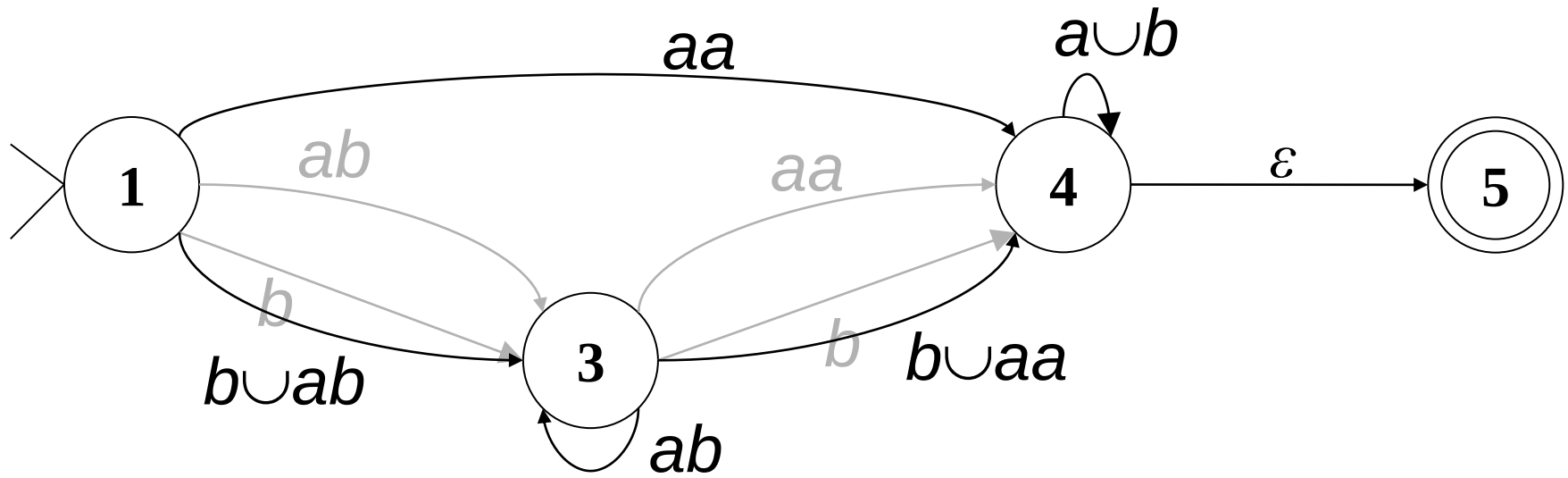
Example

- **For every pair** comprising an incoming and an outgoing edge add a new edge between its incident and outgoing states, **in that direction**.
- **For each new edge, concatenate** the incident outgoing labels, **in that order**, and decorate the new edge with that expression.



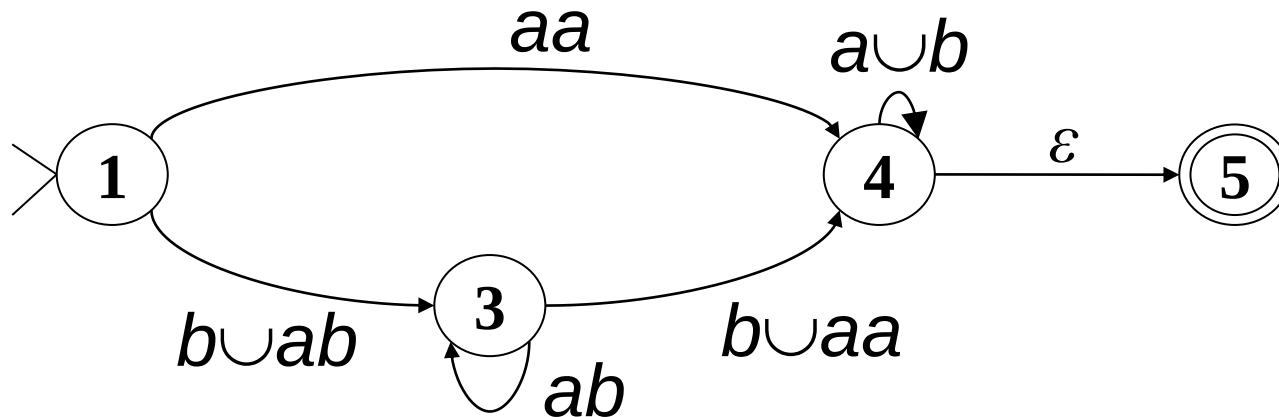
Example

- Now there are two edges from state 1 to state 3, and two from state 3 to state 4, which may be summed.

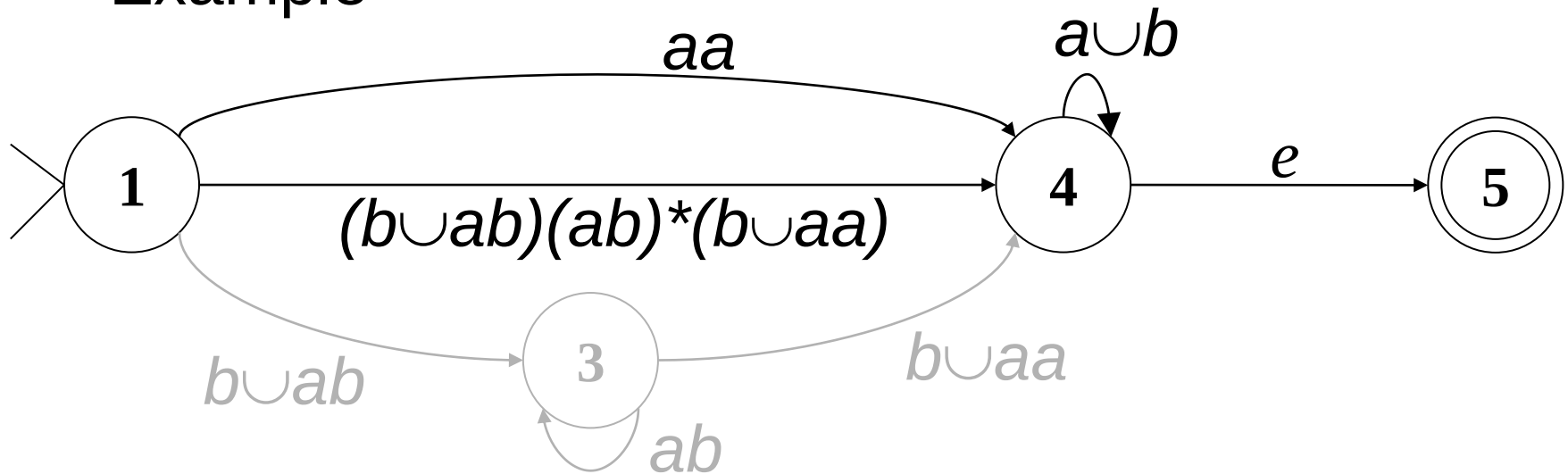


Example

- The obvious state to choose to bypass now is state 3, which has a loop on it, labelled 'ab'.
- It has one incident edge, from state 1, labelled with ' $b \cup ab$ ', and one outgoing edge, to state 4, labelled with ' $b \cup aa$ '.
- So remove it and add one new edge from state 1 to state 4 with label ' $(b \cup ab)(ab)^*(b \cup aa)$ '



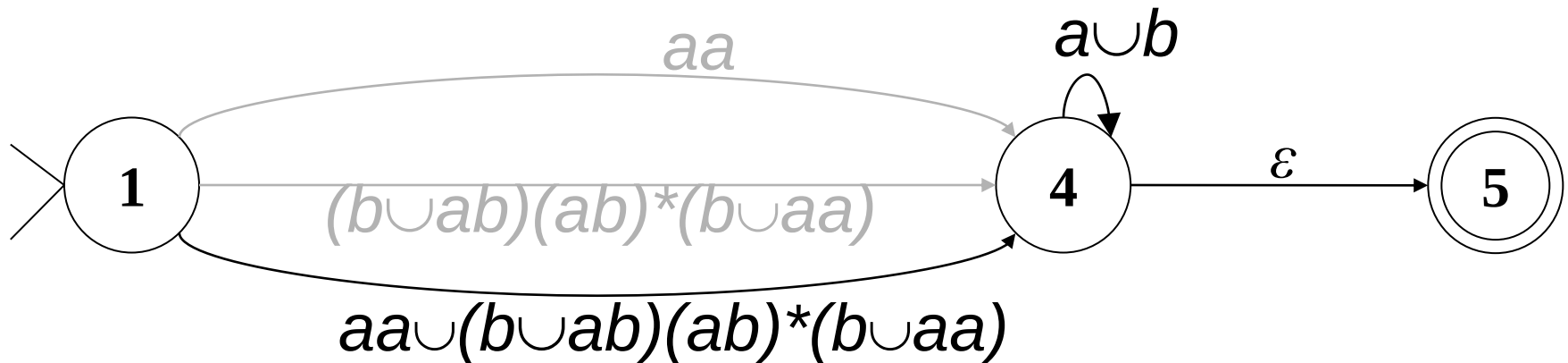
Example



- Note that each sum must be enclosed in parentheses here because concatenation ‘binds more tightly’ that summation.
- The loop term must also be enclosed in parentheses here because the Kleene Star ‘binds more tightly’ that concatenation.

Example

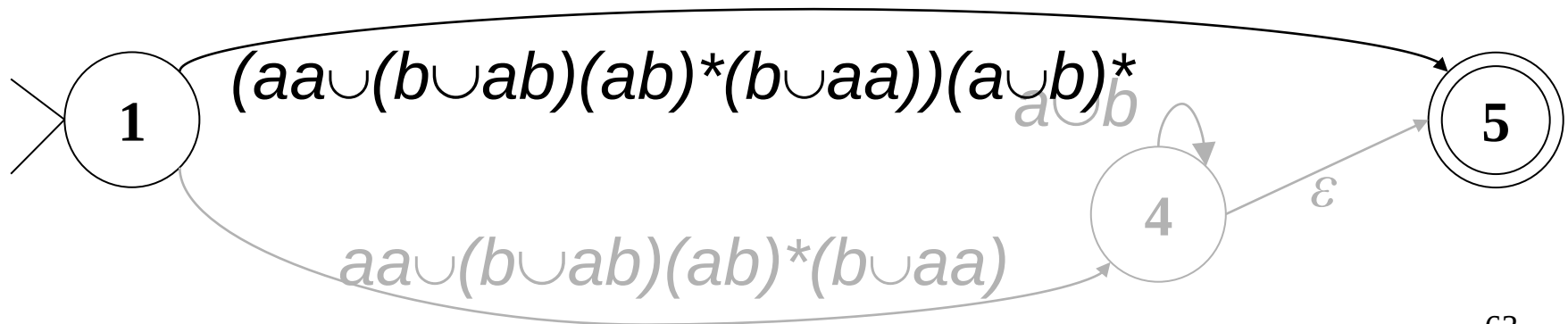
- Now there are two edges from state 1 to state 4, so sum them.



- Now there is only one choice of state to bypass: state 4.

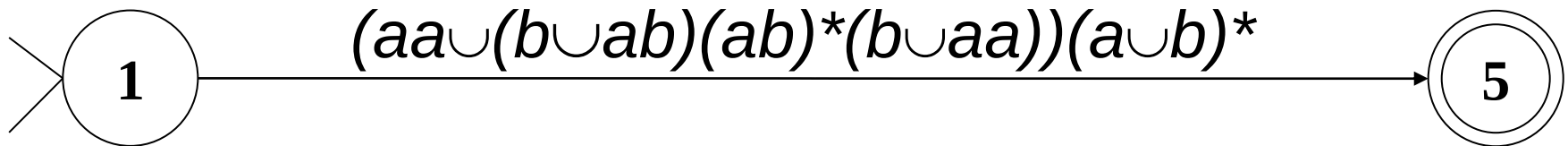
Example

- This has one incident edge, from state 1, labelled ' $aa \cup (b \cup ab)(ab)^*(b \cup aa)$ ', and one outgoing edge, to state 5, labelled ' ϵ '.
- It also has a loop on it labelled ' $a \cup b$ '.
- So removing it will break just one path, from state 1 to state 5, and the label on that path will be ' $(aa \cup (b \cup ab)(ab)^*(b \cup aa))(a \cup b)^*$ '.



Example

- Again, note the use of parentheses.
- Note also that the final ' ϵ ' is omitted because, for every RE, x , $x\epsilon = \epsilon x = x$.
- To re-iterate the final RE obtained:



Some questions

- Every regular expression is associated with some language, but is it also true that every language is associated with some regular expression? [No!]
- [But every regular language is associated with a RE: construct an FSA following the definition of a regular language, then apply Kleene's theorem.]

Summary

- Regular expressions describe the languages accepted by FSA.
- Kleene's theorem shows the equivalence of FSA and regular expressions.
- An algorithm to systematically find a regular expression for the language accepted by an FSA exists.

Reading

- Sipser:
- Pages 63-76.
- Try exercises 1.18-1.21, 1.28.